

LA-UR-21-28340

Approved for public release; distribution is unlimited.

Title: FleCSI Intro 2.0

Author(s): Bergen, Benjamin Karl

Intended for: General Information

Issued: 2021-08-20

Disclaimer:

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.



Los Alamos

NATIONAL LABORATORY

EST. 1943



Delivering science and technology
to protect our nation
and promote world stability



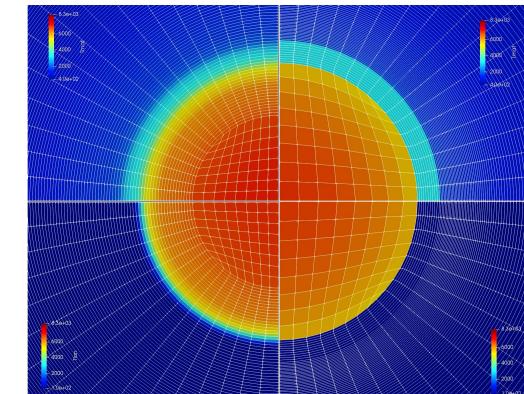
Managed by Triad National Security, LLC for the U.S. Department of Energy's NNSA

What is FleCSI?

FleCSI is a programming system or framework for developing Multiphysics application codes.

FleCSI provides two primary capabilities:

- Runtime Abstraction Layer
 - Single-source, hierarchical parallelism using **tasks** and **kernel**s
 - *Tasks* – Distributed-memory (similar role to MPI) [*Legion*, *MPI*, *HPX*]
 - *Kernels* – Shared-memory (similar role to OpenMP or CUDA) [*Kokkos*, *Kitsune*]
- Topology Data Structures
 - Customizable C++ types configurable via template parameters
 - Current topologies include: **unstructured**, **narray**, **ntree**, and **set**



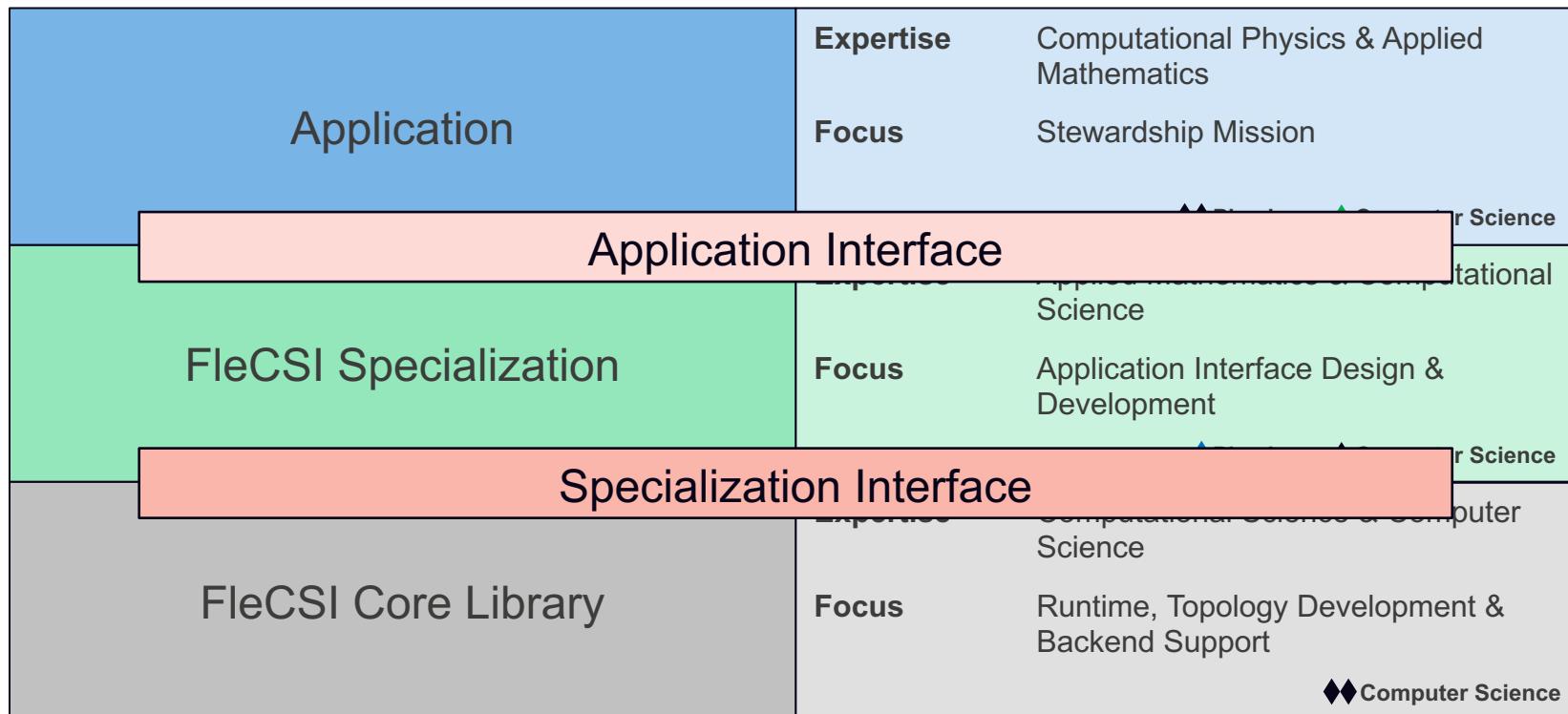
FleCSI Application Structure

FleCSI prescribes an application structure that is designed to support *separation of concerns*

Application	Expertise Computational Physics & Applied Mathematics Focus Stewardship Mission ◆ Physics ◆ Computer Science
FleCSI Specialization	Expertise Applied Mathematics & Computational Science Focus Application Interface Design & Development ◆ Physics ◆ Computer Science
FleCSI Core Library	Expertise Computational Science & Computer Science Focus Runtime, Topology Development & Backend Support ◆◆ Computer Science

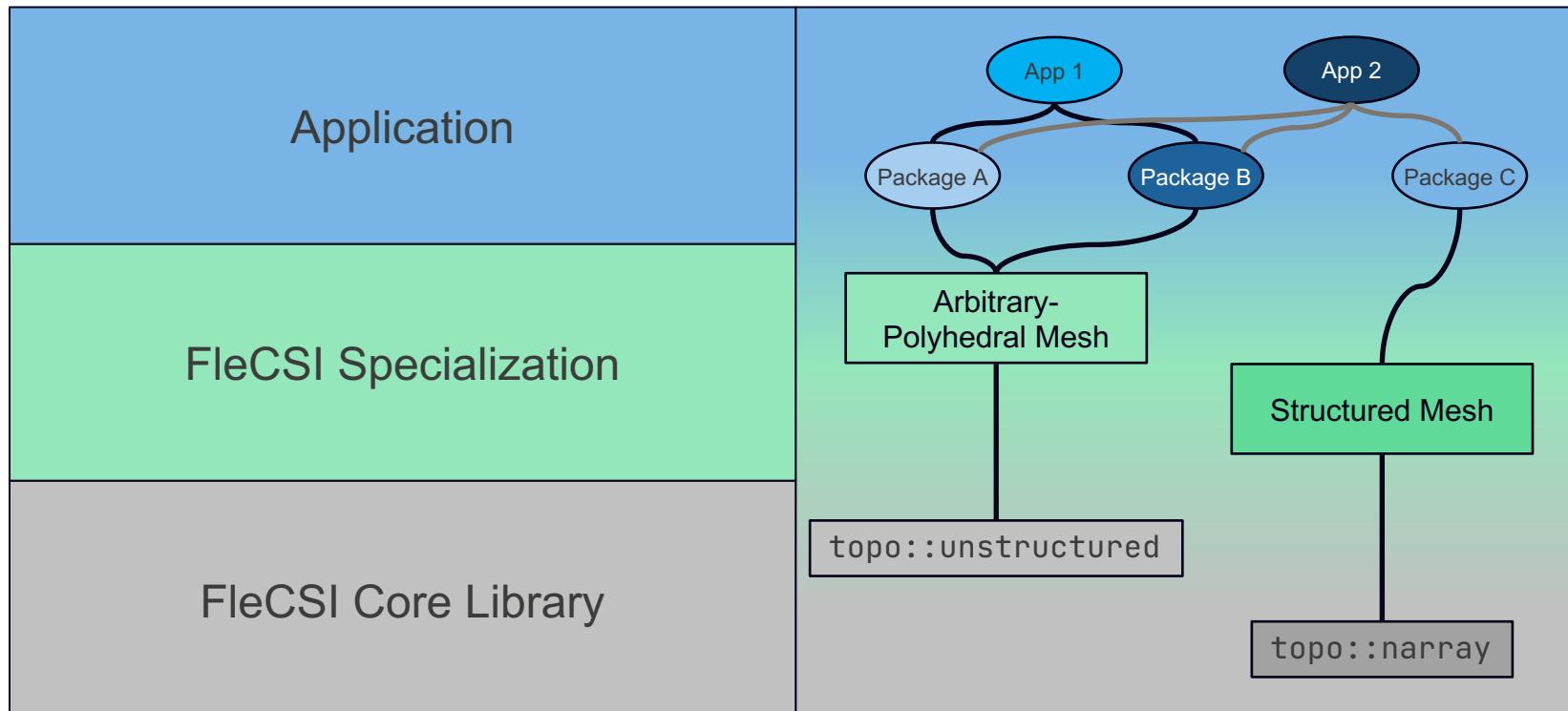
FleCSI Application Structure

FleCSI prescribes an application structure that is designed to support *separation of concerns*



FleCSI Application Structure

FleCSI prescribes an application structure that is designed to support *separation of concerns*



Things that you may find odd or irritating... (in addition to me)

- FleCSI is not like MPI.
 - Tasks share some similarities with MPI, but it is good to think of the FleCSI programming model as something really new.
- In general, you cannot use pointers.
 - The FleCSI data and execution models allow relocation of data and tasks to other nodes(address spaces), which will invalidate any pointers.
- Execution is really asynchronous!
 - Debugging can be challenging.
- It is important to understand the types and granularity of parallelism in your numerical method.

Abstraction Layer: Execution, Control & Runtime Models

Simple Task Example

```
// This is a simple task
void hello() {
    flog(info) << "Hello World" << std::endl;
}

int action() {
    // This is how you invoke a task
    execute<hello>();
    return 0;
}
```

Simple Task Example (full)

```
#include <flecsi/execution.hh>
#include <flecsi/flog.hh>

using namespace flecsi;

// This is a simple task
void hello() {
    flog(info) << "Hello World" << std::endl;
}

int action() {
    // This is how you invoke a task
    execute<hello>();
    return 0;
}

int main(int argc, char ** argv) {
    auto status = flecsi::initialize(argc, argv);

    if(status != flecsi::run::status::success) {
        return status < flecsi::run::status::clean ? 0 : status;
    }

    flecsi::log::add_output_stream("clog", std::clog, true);

    status = flecsi::start(action);

    flecsi::finalize();

    return status;
}
```

Simple Task Example (full)

Execution Model: Task

```
#include <flecsi/execution.hh>
#include <flecsi/flog.hh>

using namespace flecsi;

// This is a simple task
void hello() {
    flog(info) << "Hello World" << std::endl;
}

int action() {
    // This is how you invoke a task
    execute<hello>();
    return 0;
}

int main(int argc, char ** argv) {
    auto status = flecsi::initialize(argc, argv);

    if(status != flecsi::run::status::success) {
        return status < flecsi::run::status::clean ? 0 : status;
    }

    flecsi::log::add_output_stream("clog", std::clog, true);

    status = flecsi::start(action);

    flecsi::finalize();

    return status;
}
```

Simple Task Example (full)

Runtime Model: Runtime Control

```
#include <flecsi/execution.hh>
#include <flecsi/flog.hh>

using namespace flecsi;

// This is a simple task
void hello() {
    flog(info) << "Hello World" << std::endl;
}

int action() {
    // This is how you invoke a task
    execute<hello>();
    return 0;
}

int main(int argc, char ** argv) {
    auto status = flecsi::initialize(argc, argv);

    if(status != flecsi::run::status::success) {
        return status < flecsi::run::status::clean ? 0 : status;
    }

    flecsi::log::add_output_stream("clog", std::clog, true);

    status = flecsi::start(action);

    flecsi::finalize();

    return status;
}
```

Simple Task Example (full)

Control Model: Action

```
#include <flecsi/execution.hh>
#include <flecsi/flog.hh>

using namespace flecsi;

// This is a simple task
void hello() {
    flog(info) << "Hello World" << std::endl;
}

int action() {
    // This is how you invoke a task
    execute<hello>();
    return 0;
}

int main(int argc, char ** argv) {
    auto status = flecsi::initialize(argc, argv);

    if(status != flecsi::run::status::success) {
        return status < flecsi::run::status::clean ? 0 : status;
    }

    flecsi::log::add_output_stream("clog", std::clog, true);

    status = flecsi::start(action);

    flecsi::finalize();

    return status;
}
```

Task Example with Kernel Execution

```
using namespace flecsi;

void poisson::task::jacobi(mesh::accessor<ro> m,
    field<double>::accessor<rw, ro> ua_new,
    field<double>::accessor<ro, ro> ua,
    field<double>::accessor<ro, ro> fa,
    double omega) {
    auto u_new = m.mdslice<mesh::vertices>(ua_new);
    auto u = m.mdslice<mesh::vertices>(ua);
    auto f = m.mdslice<mesh::vertices>(fa);
    const auto dsqr = pow(m.delta(), 2);

    forall(j, m.vertices<mesh::y_axis>()) {
        forall(i, m.vertices<mesh::x_axis>()) {
            u_new[j][i] = (1.0 - omega) * u[j][i] + omega * (0.25 * (dsqr * f[j][i] +
                u[j][i + 1] + u[j][i - 1] + u[j + 1][i] + u[j - 1][i]));
        } // for
    }; // forall
} // jacobi
```

Task Example with Kernel Execution

Execution Model: Kernel

```
using namespace flecsi;

void poisson::task::jacobi(mesh::accessor<ro> m,
    field<double>::accessor<rw, ro> ua_new,
    field<double>::accessor<ro, ro> ua,
    field<double>::accessor<ro, ro> fa,
    double omega) {
    auto u_new = m.mdslice<mesh::vertices>(ua_new);
    auto u = m.mdslice<mesh::vertices>(ua);
    auto f = m.mdslice<mesh::vertices>(fa);
    const auto dsqr = pow(m.delta(), 2);

    for(auto j : m.vertices<mesh::y_axis>()) {
        forall(i, m.vertices<mesh::x_axis>()) {
            u_new[j][i] = (1.0 - omega) * u[j][i] + omega * (0.25 * (dsqr * f[j][i] +
                u[j][i + 1] + u[j][i - 1] + u[j + 1][i] + u[j - 1][i]));
        } // for
    }; // forall
} // jacobi
```

$$u_{new}[j][i] = (1 - \omega)u[j][i] + \omega(0.25(\delta^2 f[j][i] + u[j][i + 1] + u[j][i - 1] + u[j + 1][i] + u[j - 1][i]))$$

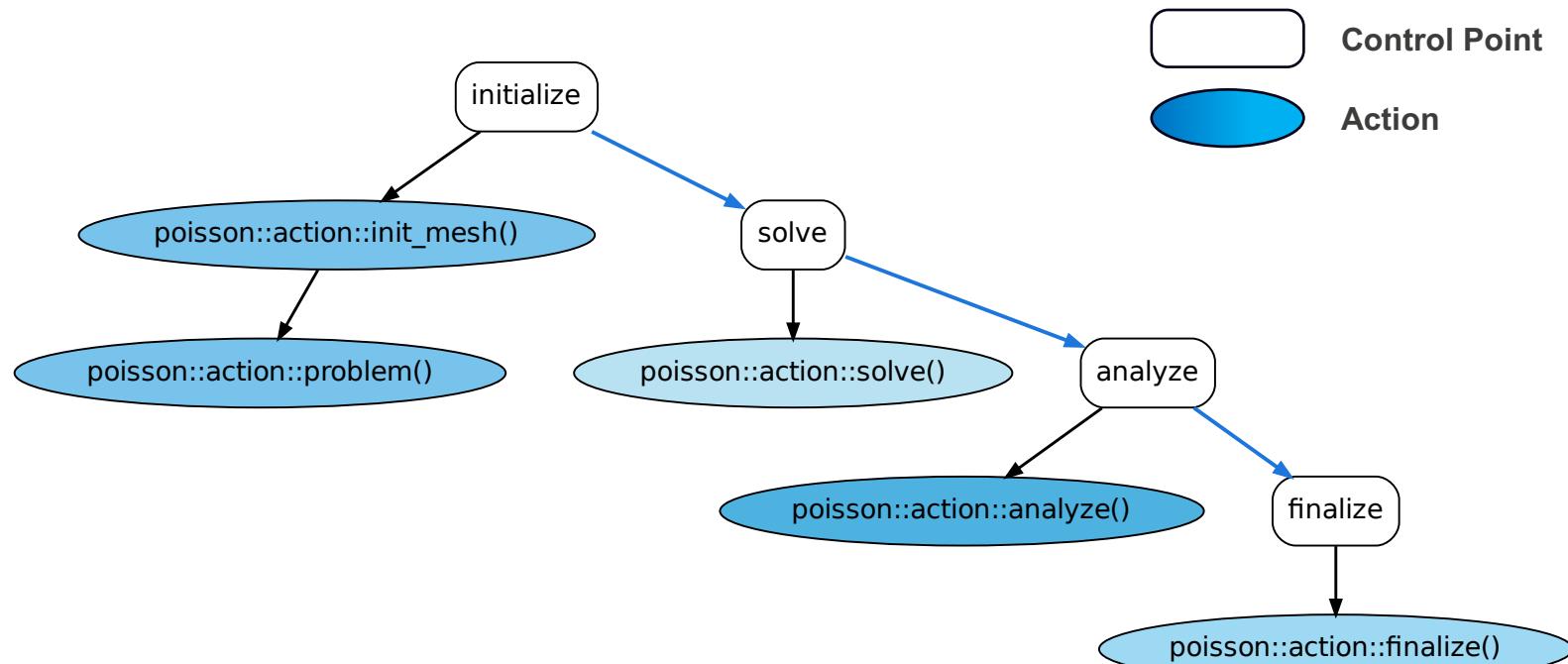
FleCSI has explicit *runtime*, *control*, *execution*, and *data models*

- **Runtime Model:** The runtime model is how you control the FleCSI runtime system itself. The basic interface includes the functions: *initialize()*, *start()*, and *finalize()*. The runtime model also provides support for creating *command-line* options, a logging utility called *flog*, and a timing and profiling interface to *Caliper*.
- **Control Model:** The control model is how the overall execution structure of the program is defined. FleCSI provides a core control type that can be customized with specialization-defined *control points* (the skeleton of the application structure). Application developers register *actions* under the control points, which may have dependencies on each other. FleCSI sorts the actions under a control point to create a runtime *program order*.
- **Execution Model:** The execution model is how work is done. FleCSI has three mechanisms for executing work: *actions* (as part of the control model), *tasks*, and *kernels*. **Actions** are C++ functions with a specific signature, but no other special characteristics. **Tasks** are *pure* functions that operate on data expressed in *regions*. These may be distributed across a collection of computing resources with different address spaces. **Kernels** act on data in a local address space in a *relaxed-consistency*, data-parallel memory model. For intuition, think of actions as normal C++ functions, tasks as C++ functions that may be executed in a different address space, and kernels as Kokkos or CUDA kernels.
- **Data Model:** The data model defines how *regions* are partitioned across the address spaces of the computing resources. FleCSI defines different *layouts* that are useful for computational physics modeling, e.g., the *dense* layout is logically an array, the *ragged* layout is a *ragged right* array, the *sparse* layout is similar to a *compressed-row storage* scheme, and the *particle* layout is a *bucket array* with support for *skip-field* iteration.

FleCSI has explicit *runtime*, *control*, *execution*, and *data models*

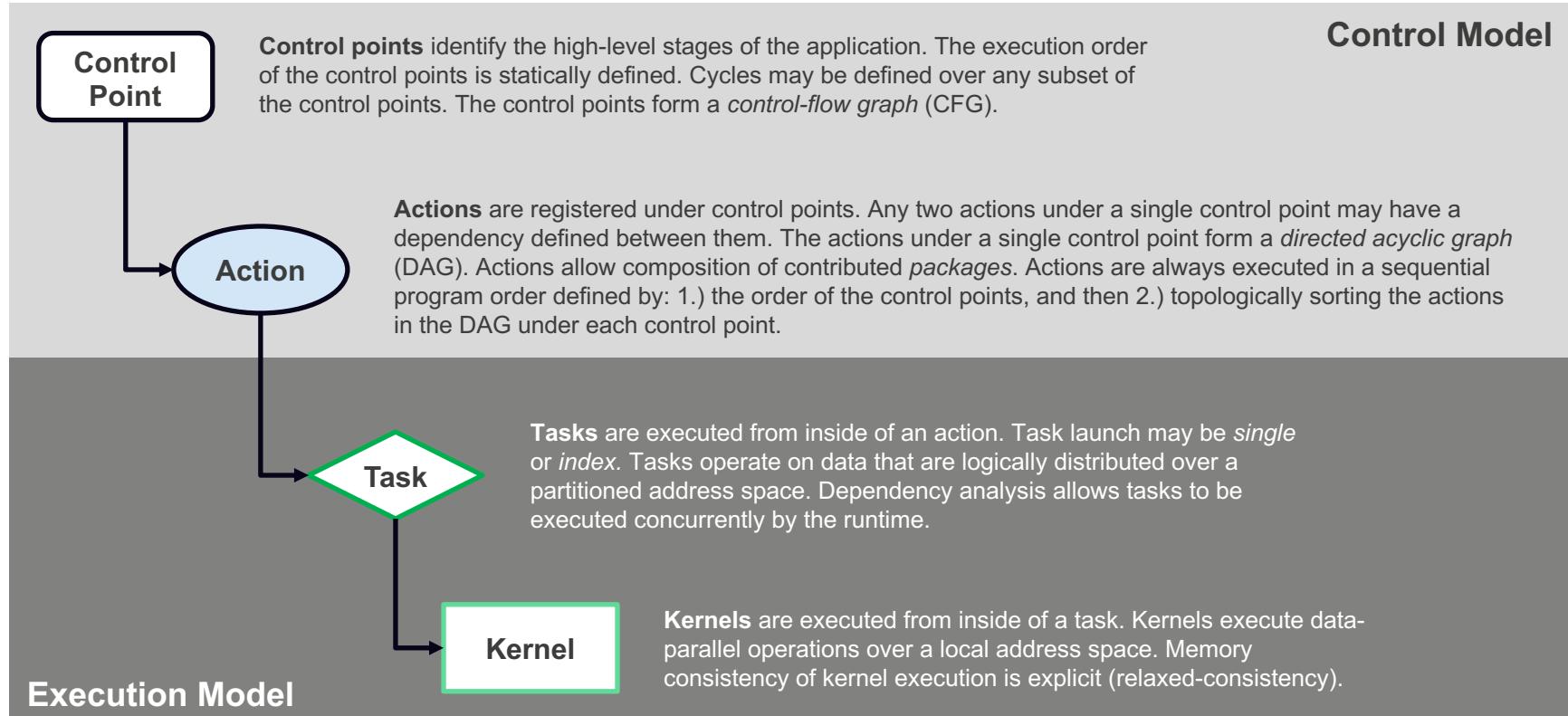
- **Runtime Model:** The runtime model is how you control the FleCSI runtime system itself. The basic interface includes the functions: *initialize()*, *start()*, and *finalize()*. The runtime model also provides support for creating *command-line* options, a logging utility called *flog*, and a timing and profiling interface to *Caliper*.
- **Control Model:** The control model is how the overall execution structure of the program is defined. FleCSI provides a core control type that can be customized with specialization-defined *control points* (the skeleton of the application structure). Application developers register *actions* under the control points, which may have dependencies on each other. FleCSI sorts the actions under a control point to create a runtime *program order*.
- **Execution Model:** The execution model is how work is done. FleCSI has three mechanisms for executing work: *actions* (as part of the control model), *tasks*, and *kernels*. **Actions** are C++ functions with a specific signature, but no other special characteristics. **Tasks** are *pure* functions that operate on data expressed in *regions*. These may be distributed across a collection of computing resources with different address spaces. **Kernels** act on data in a local address space in a *relaxed-consistency*, data-parallel memory model. For intuition, think of actions as normal C++ functions, tasks as C++ functions that may be executed in a different address space, and kernels as Kokkos or CUDA kernels.
- **Data Model:** The data model defines how *regions* are partitioned across the address spaces of the computing resources. FleCSI defines different *layouts* that are useful for computational physics modeling, e.g., the *dense* layout is logically an array, the *ragged* layout is a *ragged right* array, the *sparse* layout is similar to a *compressed-row storage* scheme, and the *particle* layout is a *bucket array* with support for *skip-field* iteration.

Control Model (*Example from Poisson Standalone App*)



- **Control Points:** defined by the specialization
- **Actions:** added by application developers

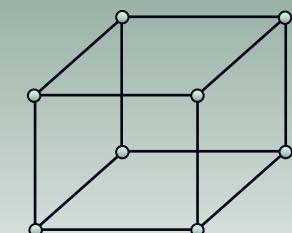
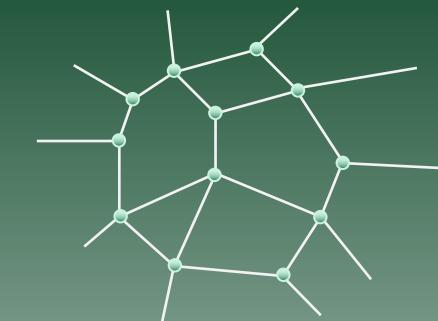
Control & Execution Models are Hierarchical



Topology Data Structures

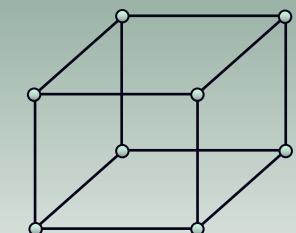
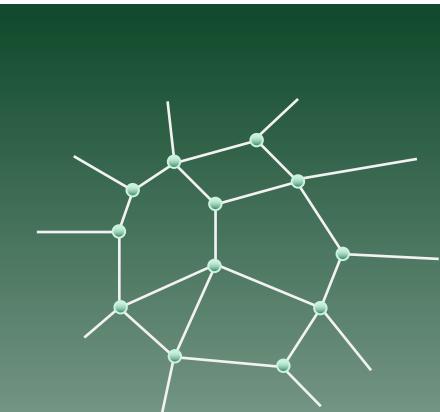
FleCSI Topologies

- **topo::unstructured**
 - Support for unstructured meshes with specialization of adjacency storage, special iterators, and user interface.
- **topo::array**
 - Support for n-dimensional arrays with specialization of user interface



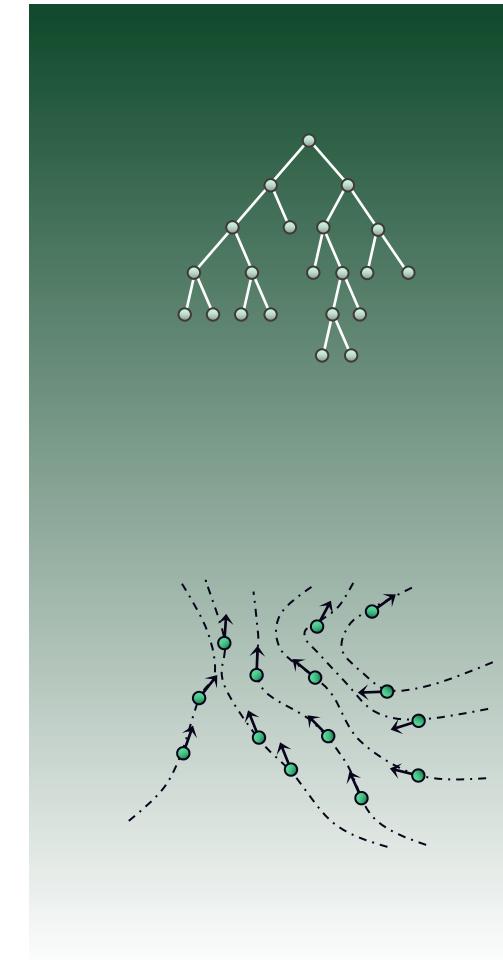
FleCSI Topologies

- **topo::unstructured**
 - Hydrodynamics (Eulerian, Lagrangian, ALE, Re-ALE, DG), Radiation/Heat Conductivity
- **topo::narray**
 - Structured mesh representations (S_N Transport, Block AMR, PIC, MPM), General n -dimensional array applications (Vlasov Problem)



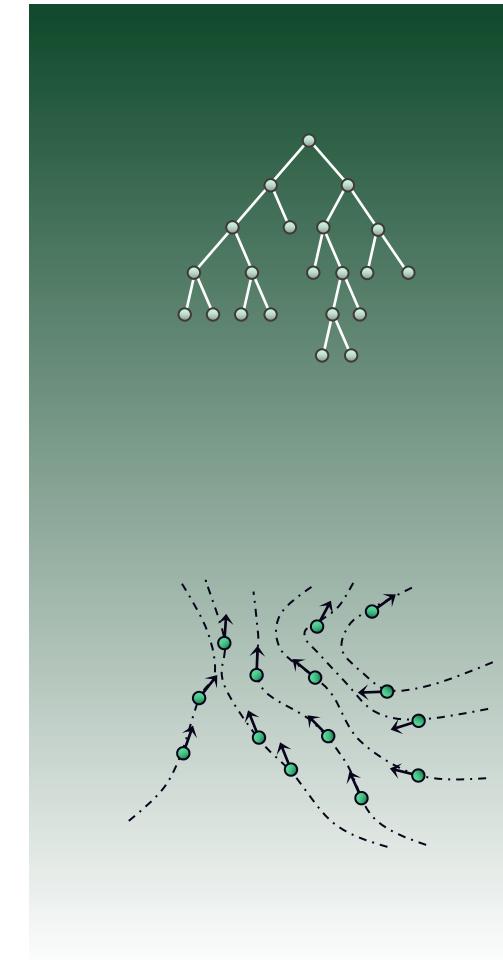
FleCSI Topologies

- **topo::ntree**
 - Support for n -dimensional hashed trees with specialization of tree types (*leaf* and *node*), neighbor definition, and user interface
- **topo::set**
 - Support for distributed sets with specialization of element type, migration rules (coloring), sorting/binning, and user interface



FleCSI Topologies

- **topo::ntree**
 - N-Body, Smoothed-Particle Hydrodynamics (SPH), Adaptive Mesh Refinement (AMR)
- **topo::set**
 - Particle-in-Cell (PIC), Material-Point Method (MPM), Charged/Neutral Particle Transport



Concepts: Index Spaces

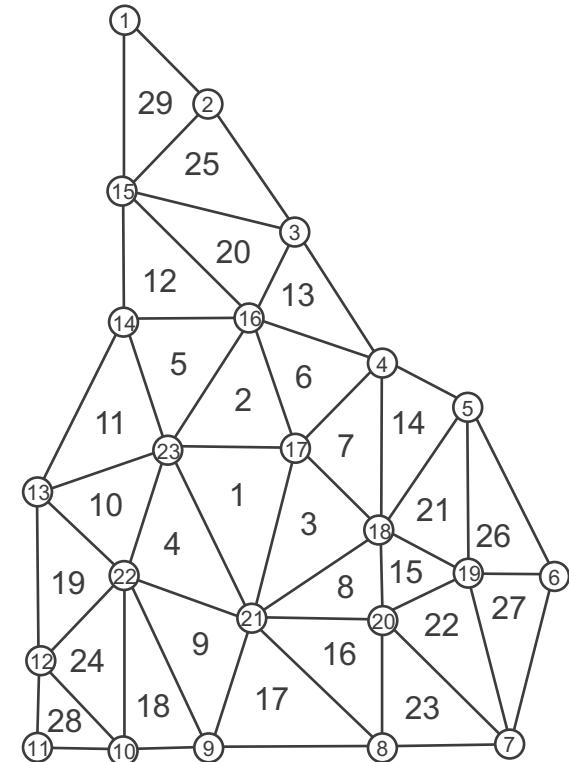
- An *index space* or *index set* is an enumerated set, e.g.,

```
for(int i=0; i<5; ++i) {  
    std::cout << "i=" << i << std::endl;  
} // for
```

- Implicitly, this for loop defines an index space of the values assumed by the loop counter i :
 $\{ 0, 1, 2, 3, 4 \}$

Concepts: Index Spaces

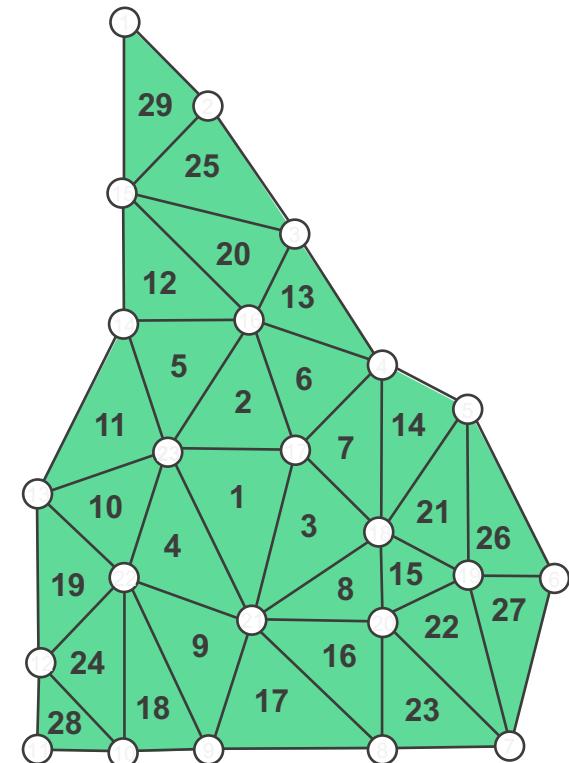
A more relevant index space example is the set of indices for a canonical triangle mesh



Concepts: Index Spaces

A more relevant index space example is the set of indices for a canonical triangle mesh

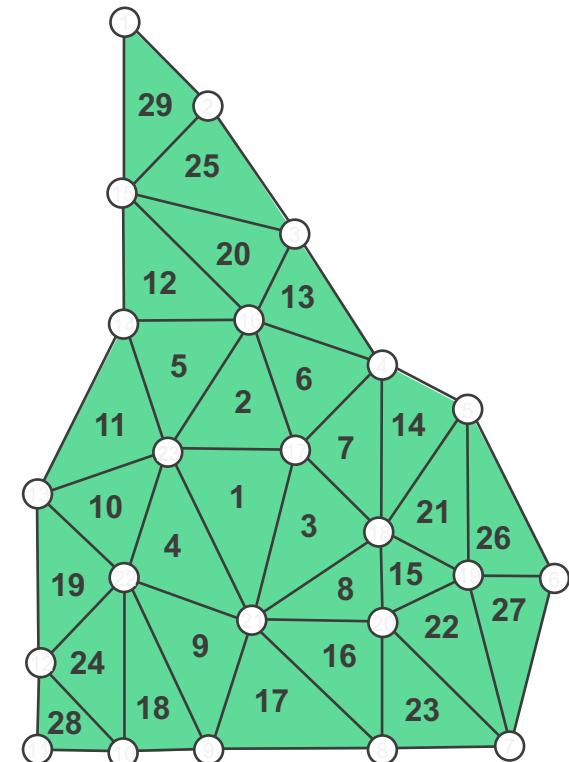
- The cells index space has 29 objects, indexed from 1 to 29
 - { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 }



Concepts: Index Spaces

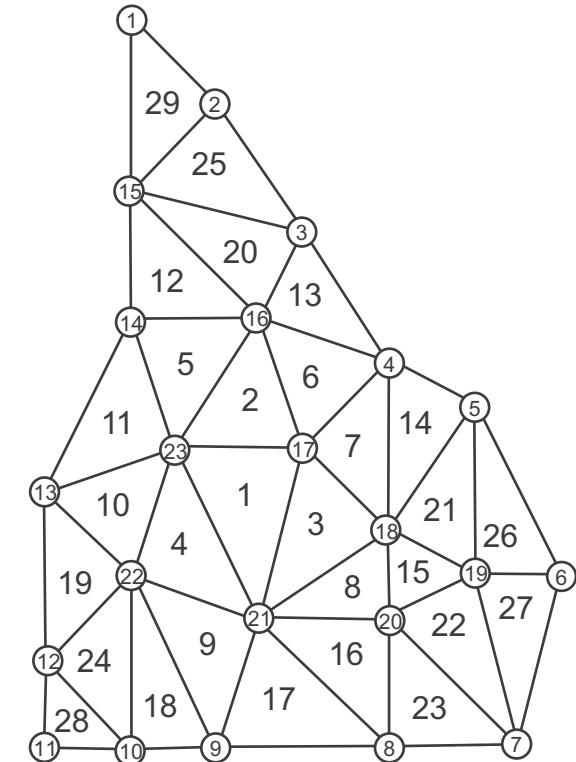
A more relevant index space example is the set of indices for a canonical triangle mesh

- The cells index space has 29 objects, indexed from 1 to 29
 - { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 }
- An index space defines the number and association of data, e.g., pressure could be defined at each cell center



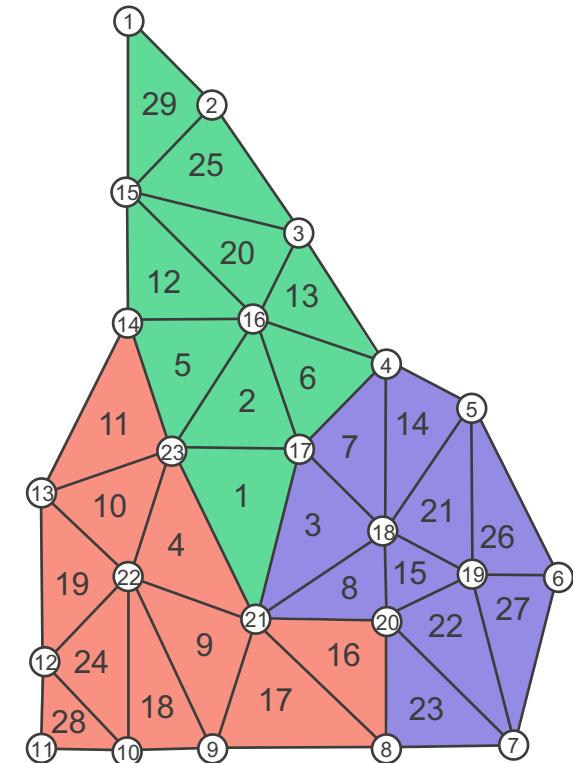
Concepts: FleCSI Distributed Mesh Model

FleCSI uses a fairly standard approach to distributing meshes



Concepts: FleCSI Distributed Mesh Model

An input mesh is partitioned using an appropriate *coloring* algorithm (chosen by the user), e.g., ParMETIS k-way graph partitioning to reduce edge cuts

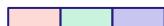


Concepts: FleCSI Distributed Mesh Model

Each color is then *closed* with respect to its dependencies, defining *exclusive*, *shared* and *ghost* entities



Exclusive: I own them, and nobody else depends on them directly



Shared: I own them, and others depend on them directly



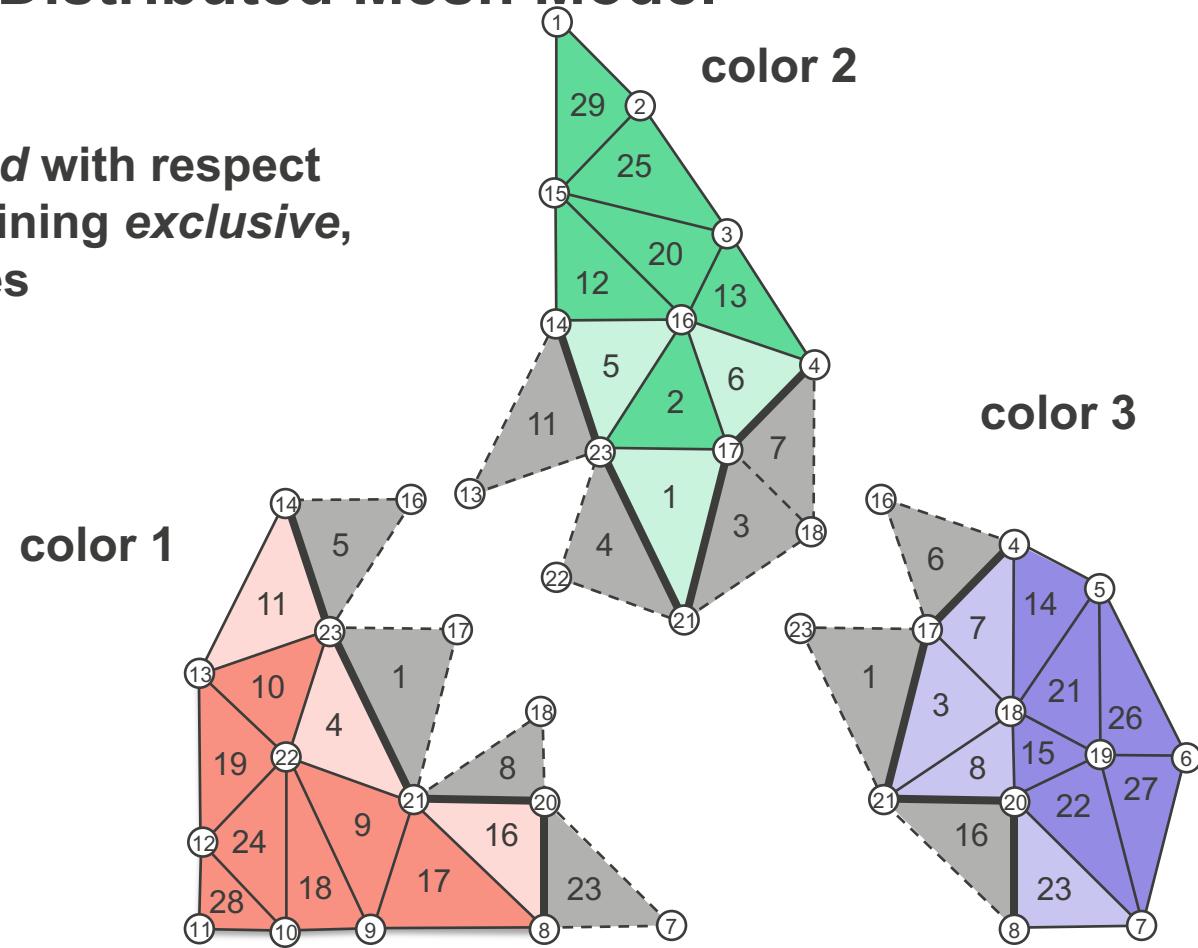
Ghost: Somebody else owns them, and I depend on them directly



Owned: Exclusive + Shared

Intuitions:

- My shared is someone else's ghost
- Exclusive, shared and ghost are index subspaces



Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

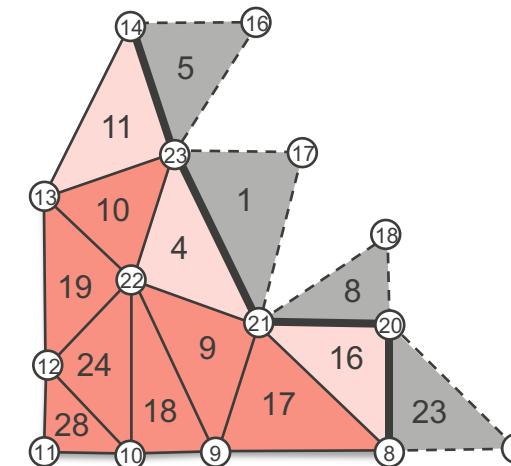
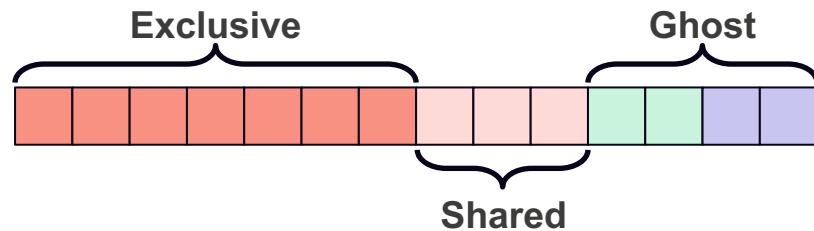
void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

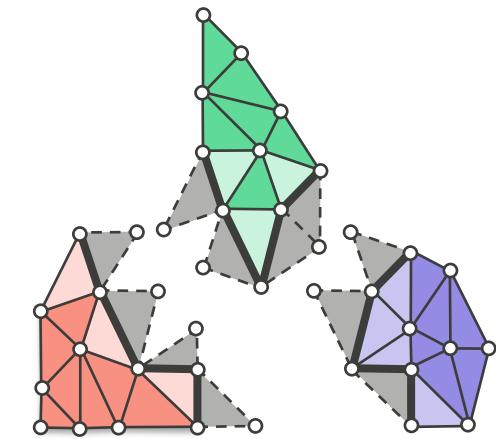
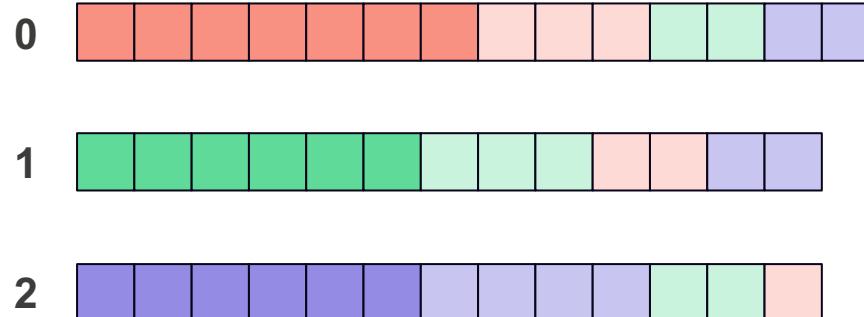
void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

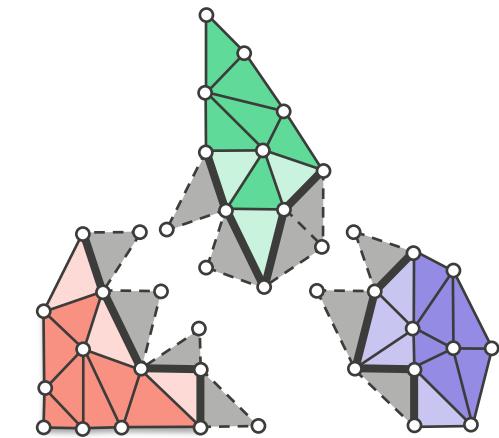
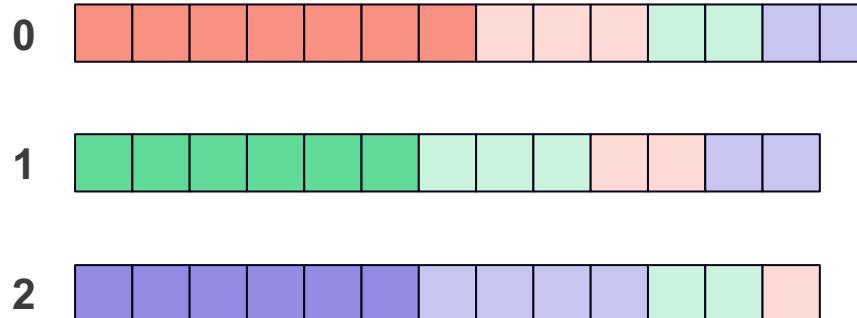
FleCSI Data Privileges



FleCSI Data Privileges

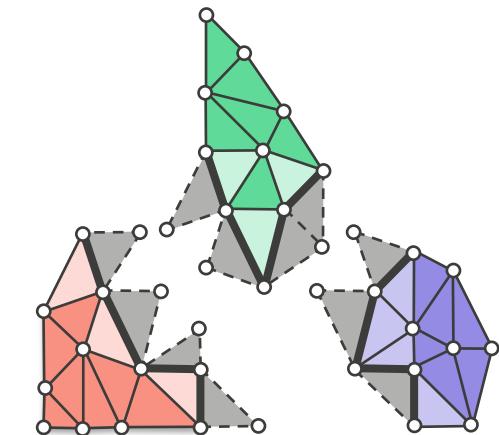
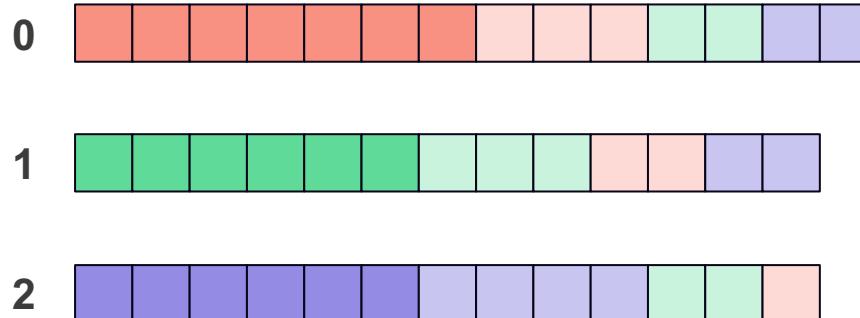


FleCSI Data Privileges



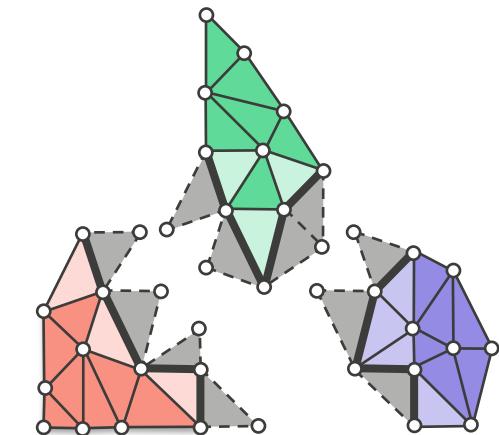
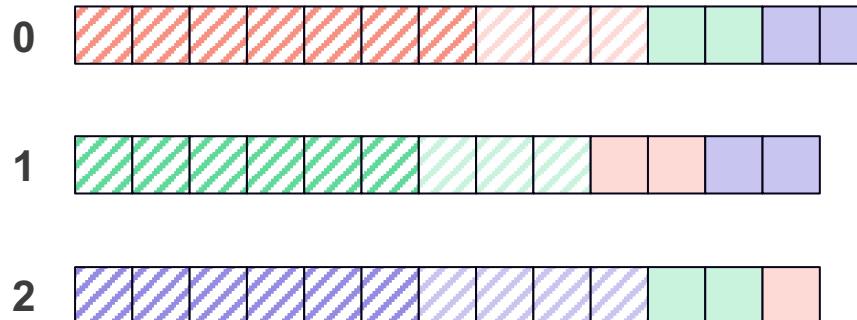
```
void taskOne(mesh<ro> m, field<rw, rw, ro> f)
```

FleCSI Data Privileges



```
void taskOne(mesh<ro> m, field<rw, ro> f)
```

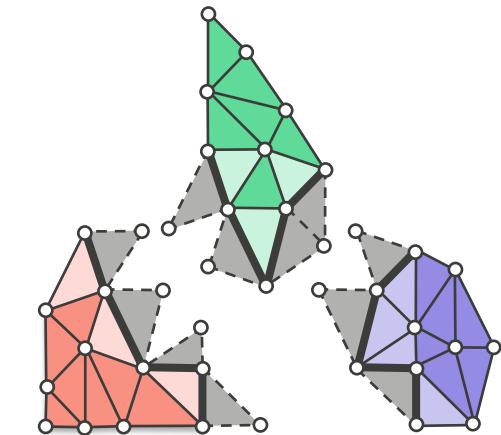
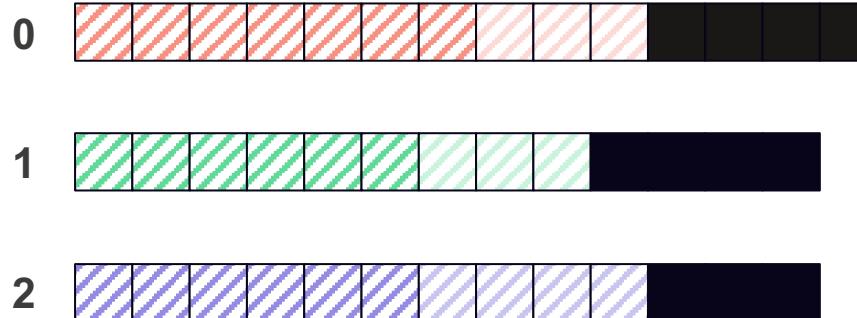
FleCSI Data Privileges



```
void taskOne(mesh<ro> m, field<rw, ro> f)
```

Read-write access on shared values allows update of data

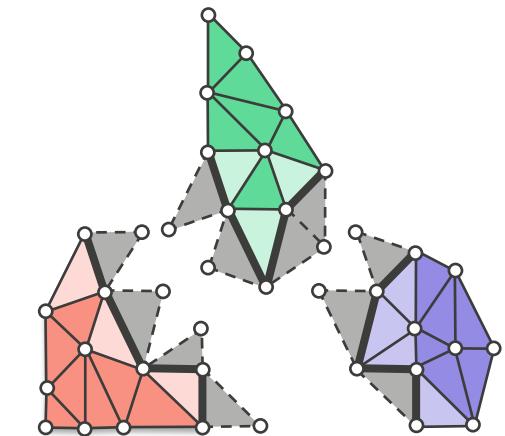
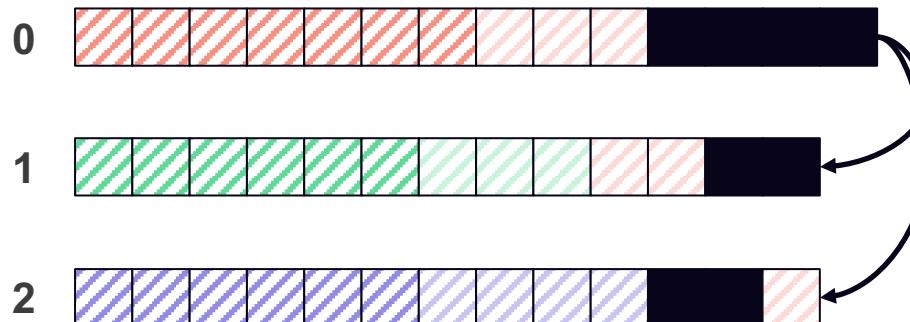
FleCSI Data Privileges



```
void taskOne(mesh<ro> m, field<rw, ro> f)
```

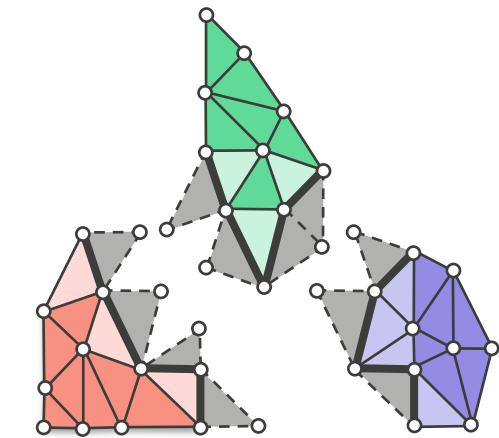
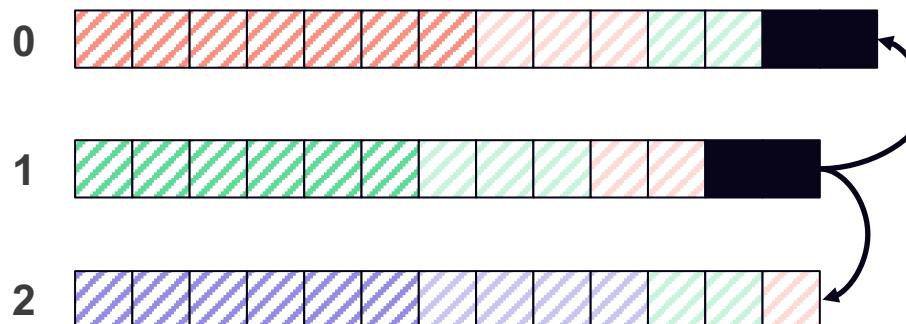
Execution of this task invalidates the read-only ghosts

FleCSI Data Privileges



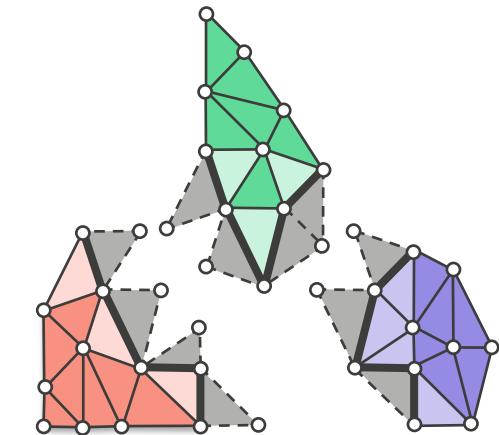
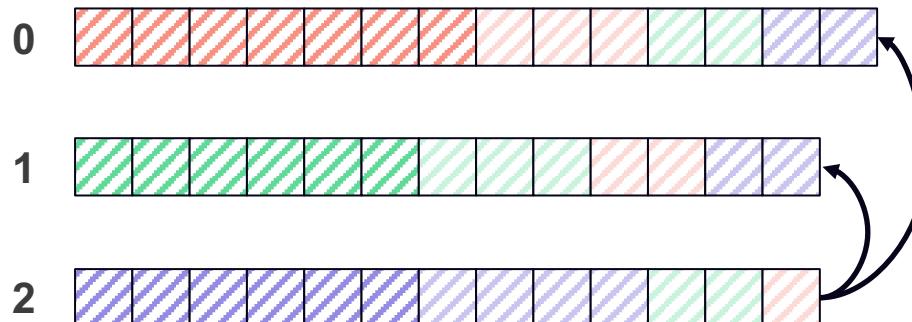
Data exchange updates distributed-memory consistency

FleCSI Data Privileges



Data exchange updates distributed-memory consistency

FleCSI Data Privileges



Data exchange updates distributed-memory consistency

Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

Field Example

```
inline const field<double>::definition<mesh, mesh::cells> pd; /* pressure definition */

void init_pressure(mesh::accessor<ro> m, field<double>::accessor<wo, wo, na> p) {
    for(auto c: m.cells()) {
        p[c] = double(color());
    }
}

void update_pressure(mesh::accessor<ro> m, field<double>::accessor<rw, rw, ro> p) {
    for(auto c: m.cells()) {
        p[c] += 1.0;
    }
}

void print_pressure(mesh::accessor<ro> m, field<double>::accessor<ro, ro, ro> p) {
    std::stringstream ss;
    ss << "pressure" << std::endl;
    for(auto c: m.cells()) {
        ss << p[c] << " ";
    }
    flog(info) << ss.str() << std::endl;
}

int example_action() {
    // By hook or by crook, we have `m` an instance of the mesh specialization
    execute<init_pressure>(p(m));
    execute<update_pressure>(p(m));
    execute<print_pressure>(p(m));
    return 0;
}
```

FleCSI Data Privileges

User-specified privileges on task arguments allow FleCSI to maintain data coherence at task edges.

na No–Access (defers consistency update)

ro Read-Only

wo Write-Only (overwrite current values)

rw Read-Write

Write privileges on ghosts have special behavior

Exclusive	Shared	Ghost
-----------	--------	-------

x	rw/wo	ro	shared state is invalidated → exchange
---	-------	----	--

x	x	rw/wo	ghosts updated by user are valid → no exchange
---	---	-------	--

Resources

Main Project Repository	<code>gitlab.lanl.gov:flecsi/flecsi.git</code>
Sphinx Documentation	<code>flecsi.org</code>
Build Instructions	<code>https://gitlab.lanl.gov/flecsi/flecsi/-/wikis/Darwin-Build</code>

The best place to start looking at code is the *feature* branch 2:

```
$ git clone --single-branch --branch 2 git@gitlab.lanl.gov:flecsi/flecsi.git  
$ cd flecsi/tutorial
```